services

Services Overview ■ Services in the PlayPlus architecture handle data operations, business logic, and external API communication. They provide a clean abstraction layer between your application and external data sources, ensuring consistent data management across your application. Key Characteristics ■ Data Layer Abstraction ■ Handle all external API communication Provide consistent data interfaces Manage caching and state synchronization Handle error scenarios gracefully State Management ■ Use RxJS BehaviorSubject for reactive state Provide observable streams for data updates Implement caching strategies Support real-time data synchronization Error Handling ■ Comprehensive error handling Retry mechanisms User-friendly error messages Logging and monitoring support Template Structure ■ TypeScript Service (service.ts.hbs) ■ import { Injectable } from "@angular/core" ; import { HttpClient } from "@angular/common/http" ; import { Observable , BehaviorSubject } from "rxjs" ; import { map , catchError } from "rxjs/operators" ; export interface ExampleData { id : string ; name : string ; // Add more properties as needed } @ Injectable ({ providedIn : "root" , }) export class ExampleService { private apiUrl = "/api/example" ; // Update with your API endpoint private dataSubject = new BehaviorSubject < ExampleData [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; constructor (private http : HttpClient) { } /** * Get all example data */ getData ( ) : Observable < ExampleData [ ] > { return this . http . get < ExampleData [ ] > (this . apiUrl) . pipe (map ((data) => { this . dataSubject . next (data) ; return data ; }) , catchError ((error) => { console . error ("Error fetching example data:" , error) ; throw error ; })) ; } /** * Get example by ID */ getById (id : string) : Observable < ExampleData > { return this . http . get < ExampleData > (` ${ this . apiUrl } / ${ id } `) . pipe (catchError ((error) => { console . error (` Error fetching example with ID ${ id } : ` , error) ; throw error ; })) ; } /** * Create new example */ create (data : Omit < ExampleData , "id" >) : Observable < ExampleData > { return this . http . post < ExampleData > (this . apiUrl , data) . pipe (map ((newData) => { const currentData = this . dataSubject . value ; this . dataSubject . next ([... currentData , newData]) ; return newData ; }) , catchError ((error) => { console . error ("Error creating example:" , error) ; throw error ; })) ; } /** * Update existing example */ update (id : string , data : Partial < ExampleData >) : Observable < ExampleData > { return this . http . put < ExampleData > (` ${ this . apiUrl } / ${ id } ` , data) . pipe (map ((updatedData) => { const currentData = this . dataSubject . value ; const updatedIndex = currentData . findIndex ((item) => item . id === id) ; if (updatedIndex !== - 1) { currentData [updatedIndex] = updatedData ; this . dataSubject . next ([... currentData]) ; } return updatedData ; }) , catchError ((error) => { console . error (` Error updating example with ID ${ id } : ` , error) ; throw error ; })) ; } /** * Delete example */ delete (id : string) : Observable < void > { return this . http . delete < void > (` ${ this . apiUrl } / ${ id } `) . pipe (map (( ) => { const currentData = this . dataSubject . value ; const filteredData = currentData . filter ((item) => item . id !== id) ; this . dataSubject . next (filteredData) ; }) , catchError ((error) => { console . error (` Error deleting example with ID ${ id } : ` , error) ; throw error ; })) ; } /** * Get current data from cache */ getCurrentData ( ) : ExampleData [ ] { return this . dataSubject . value ; } /** * Clear cached data */ clearCache ( ) : void { this . dataSubject . next ([ ]) ; } } Features ■ Built-in Features ■ CRUD Operations Create, Read, Update, Delete operations Type-safe interfaces Consistent error handling Reactive State Management BehaviorSubject for current state Observable streams for updates Automatic cache updates Error Handling Comprehensive error catching Detailed error logging Graceful error propagation Caching In-memory data caching Cache invalidation strategies Cache clearing methods Type Safety Strongly typed interfaces Generic type support IntelliSense support Best Practices ■ Service Structure @ Injectable ({ providedIn : "root" , }) export class ExampleService { private apiUrl = "/api/example" ; private dataSubject = new BehaviorSubject < ExampleData [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; } Error Handling getData ( ) : Observable < ExampleData [ ] > { return this . http . get < ExampleData [ ] > (this . apiUrl) . pipe (map (data => { this . dataSubject . next (data) ; return data ; }) , catchError (error => { console . error ('Error fetching data:' , error) ; throw error ; })) ; } Cache Management getCurrentData ( ) : ExampleData [ ] { return this . dataSubject . value ; } clearCache ( ) : void { this . dataSubject . next ([ ]) ; } Usage Examples

■ Basic Service ■ # Generate a basic service playplus generate service user This creates: UserService with CRUD operations UserData interface Comprehensive error handling Caching mechanisms Advanced Service ■ # Generate with custom interface playplus generate service product --interface=Product This creates: ProductService with custom interface Advanced caching strategies Real-time data synchronization Integration with Components ■ Container Components ■ // In container component export class UserListContainerComponent { private userService = inject (UserService) ; protected users = signal < User [ ] > ([ ]) ; constructor ( ) { this . loadUsers ( ) ; } private loadUsers ( ) : void { this . userService . getData ( ) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : (users) => this . users . set (users) , error : (error) => console . error ("Failed to load users:" , error) , }) ; } protected createUser (userData : Omit < User , "id" >) : void { this . userService . create (userData) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : (newUser) => { // User created successfully this . loadUsers ( ) ; // Refresh the list } , error : (error) => console . error ("Failed to create user:" , error) , }) ; } } Multiple Services ■ // In complex container component export class DashboardContainerComponent { private userService = inject (UserService) ; private productService = inject (ProductService) ; private orderService = inject (OrderService) ; protected users = signal < User [ ] > ([ ]) ; protected products = signal < Product [ ] > ([ ]) ; protected orders = signal < Order [ ] > ([ ]) ; constructor ( ) { this . loadDashboardData ( ) ; } private loadDashboardData ( ) : void { // Load all data in parallel combineLatest ([this . userService . getData ( ) , this . productService . getData ( ) , this . orderService . getData ( ) ,]) . pipe (takeUntilDestroyed ( )) . subscribe ({ next : ([users , products , orders]) => { this . users . set (users) ; this . products . set (products) ; this . orders . set (orders) ; } , error : (error) => console . error ("Failed to load dashboard data:" , error) , }) ; } } Data Interfaces ■ Basic Interface ■ export interface UserData { id : string ; name : string ; email : string ; isActive : boolean ; createdAt : Date ; updatedAt : Date ; } Complex Interface ■ export interface ProductData { id : string ; name : string ; description : string ; price : number ; category : ProductCategory ; images : string [ ] ; specifications : Record < string , any > ; isAvailable : boolean ; stockQuantity : number ; createdAt : Date ; updatedAt : Date ; } export enum ProductCategory { ELECTRONICS = "electronics" , CLOTHING = "clothing" , BOOKS = "books" , HOME = "home" , } Advanced Patterns ■ Caching Strategies ■ export class AdvancedExampleService { private cache = new Map < string , { data : any ; timestamp : number } > ( ) ; private readonly CACHE_DURATION = 5 * 60 * 1000 ; // 5 minutes getDataWithCache (id : string) : Observable < ExampleData > { const cached = this . cache . get (id) ; const now = Date . now ( ) ; if (cached && now - cached . timestamp < this . CACHE_DURATION) { return of (cached . data) ; } return this . http . get < ExampleData > (` ${ this . apiUrl } / ${ id } `) . pipe (map ((data) => { this . cache . set (id , { data , timestamp : now }) ; return data ; })) ; } invalidateCache (id ? : string) : void { if (id) { this . cache . delete (id) ; } else { this . cache . clear ( ) ; } } } Real-time Updates ■ export class RealTimeExampleService { private dataSubject = new BehaviorSubject < ExampleData [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; // WebSocket connection for real-time updates private wsConnection : WebSocket ; constructor ( ) { this . initializeWebSocket ( ) ; } private initializeWebSocket ( ) : void { this . wsConnection = new WebSocket ("ws://api.example.com/updates") ; this . wsConnection . onmessage = (event) => { const update = JSON . parse (event . data) ; this . handleRealTimeUpdate (update) ; } ; } private handleRealTimeUpdate (update : any) : void { const currentData = this . dataSubject . value ; switch (update . type) { case "CREATE" : this . dataSubject . next ([... currentData , update . data]) ; break ; case "UPDATE" : const updatedData = currentData . map ((item) => item . id === update . data . id ? update . data : item) ; this . dataSubject . next (updatedData) ; break ; case "DELETE" : const filteredData = currentData . filter ((item) => item . id !== update . data . id) ; this . dataSubject . next (filteredData) ; break ; } } } Testing ■ Service Testing ■ describe ("UserService" , ( ) => { let service : UserService ; let httpMock : HttpTestingController ; beforeEach (( ) => { TestBed . configureTestingModule ({ imports : [HttpClientTestingModule] , providers : [UserService] , }) ; service = TestBed . inject (UserService) ; httpMock = TestBed . inject (HttpTestingController) ; }) ; afterEach (( ) => { httpMock . verify ( ) ; }) ; it ("should fetch users successfully" , ( ) => { const mockUsers : UserData [ ] = [{ id : "1" , name : "John Doe" , email : "john@example.com" , isActive : true , createdAt : new Date ( ) , updatedAt : new Date ( ) , } ,] ; service . getData ( ) . subscribe ((users) => {

expect (users) . toEqual (mockUsers) ; }) ; const req = httpMock . expectOne ("/api/user") ; expect (req . request . method) . toBe ("GET") ; req . flush (mockUsers) ; }) ; it ("should handle errors gracefully" , ( ) => { service . getData ( ) . subscribe ({ next : ( ) => fail ("Should not succeed") , error : (error) => { expect (error . status) . toBe (500) ; } , }) ; const req = httpMock . expectOne ("/api/user") ; req . flush ("Server error" , { status : 500 , statusText : "Internal Server Error" , }) ; }) ; }) ; Error Handling Strategies ■ Retry Logic ■ import { retry , delay } from 'rxjs/operators' ; getDataWithRetry ( ) : Observable < ExampleData [ ] > { return this . http . get < ExampleData [ ] > (this . apiUrl) . pipe (retry (3) , // Retry 3 times delay (1000) , // Wait 1 second between retries catchError (error => { console . error ('Failed after retries:' , error) ; throw error ; })) ; } Error Transformation ■ import { throwError } from 'rxjs' ; private handleError (error : any) : Observable < never > { let errorMessage = 'An unknown error occurred' ; if (error . status === 404) { errorMessage = 'Resource not found' ; } else if (error . status === 500) { errorMessage = 'Server error occurred' ; } else if (error . status === 0) { errorMessage = 'Network error - please check your connection' ; } console . error ('Service error:' , error) ; return throwError (( ) => new Error (errorMessage)) ; } Performance Optimization ■ Lazy Loading ■ export class LazyExampleService { private dataSubject = new BehaviorSubject < ExampleData [ ] > ([ ]) ; public data$ = this . dataSubject . asObservable ( ) ; private loaded = false ; getData ( ) : Observable < ExampleData [ ] > { if (! this . loaded) { this . loadData ( ) ; } return this . data$; } private loadData ( ) : void { this . loaded = true ; this . http . get < ExampleData [ ] > (this . apiUrl) . pipe (map ((data) => { this . dataSubject . next (data) ; return data ; }) , catchError ((error) => { this . loaded = false ; // Allow retry throw error ; })) . subscribe ( ) ; } } Next Steps ■ After creating a service: Define the data interface for your specific domain Implement CRUD operations for your API endpoints Add error handling specific to your use case Implement caching strategies if needed Write comprehensive tests for all operations Integrate with container components for data flow Developer Checklist ■ Before Creating Services: ■ Are data interfaces defined for all entities? Is HTTP error handling implemented with meaningful messages? Is BehaviorSubject used for reactive state management? Are CRUD operations implemented for all entities? Is retry logic added for network failures? Are caching strategies implemented where needed? Are all API endpoints typed with TypeScript? Is data validation and sanitization implemented? Do I have unit tests with HTTP mocking? Is logging added for debugging and monitoring? Are RxJS operators used for async operations? Are cache invalidation strategies implemented?